The LLVM MIPS and ARM back-end

Embedded Linux Conference 2009
Grenoble, France

Bruno Cardoso Lopes
bruno.cardoso@gmail.com

Agenda

- What's LLVM?
- Why?
- LLVM design
- CodeGen
- The MIPS and ARM back-end

What's LLVM ?

Basics

- Low Level Virtual Machine
- A virtual instruction set
- A compiler infrastructure suite with aggressive optimizations.

A virtual instruction set

- Low-level representation, but with high-level type information
- 3-address-code-like representation
- RISC based, language independent with SSA information. The on-disk representation is called **bitcode**.

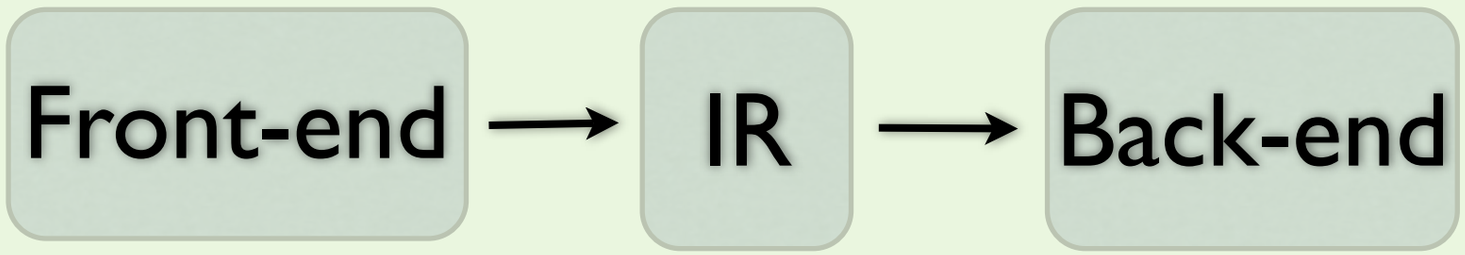
A virtual instruction set

```
int dummy(int a) {
  return a+3;
}
```

```
define i32 @dummy(i32 %a) nounwind readnone {
entry:
    %0 = add i32 %a, 3
    ret i32 %0
}
```

A compiler infrastructure suite

- Compiler front-end
  - llvm-gcc
  - clang
- IR and tools to handle it
- JIT and static back-ends.



Set of tools

- llc - invoke static back-ends.
- lli - bitcode interpreter, use JIT
- bugpoint - reduce code from crashes
- opt - run optimizations on bitcodes
- llvm-extract - extract/delete functions and data
- llvm-dis, llvm-as, llvm-ld, ...

Front-end : llvm-gcc

- GCC based front-end : llvm-gcc
- GIMPLE to LLVM IR
- GCC is not modular (intentionally)

Front-end : llvm-gcc

- Very mature and supports Java, Ada, FORTRAN, C, C++ and ObjC.
- Cross-compiler needed for a not native target.

```
$ llvm-gcc -O2 -c clown.c -emit-llvm -o clown.bc
$ llvm-extract -func=bozo < clown.bc | llvm-dis

define float @bozo(i32 %lhs, i32 %rhs, float %w) nounwind  {
entry:
    %0 = sdiv i32 %lhs, %rhs
    %1 = sitofp i32 %0 to float
    %2 = mul float %1, %w
    ret float %2
}
```

LLVM assembly

Front-end : clang

- Clang: C, ObjC front-end.
- C++ in progress
- Better diagnostics
- Integration with IDEs
- No need to generate a cross-compiler
- Static Analyzer

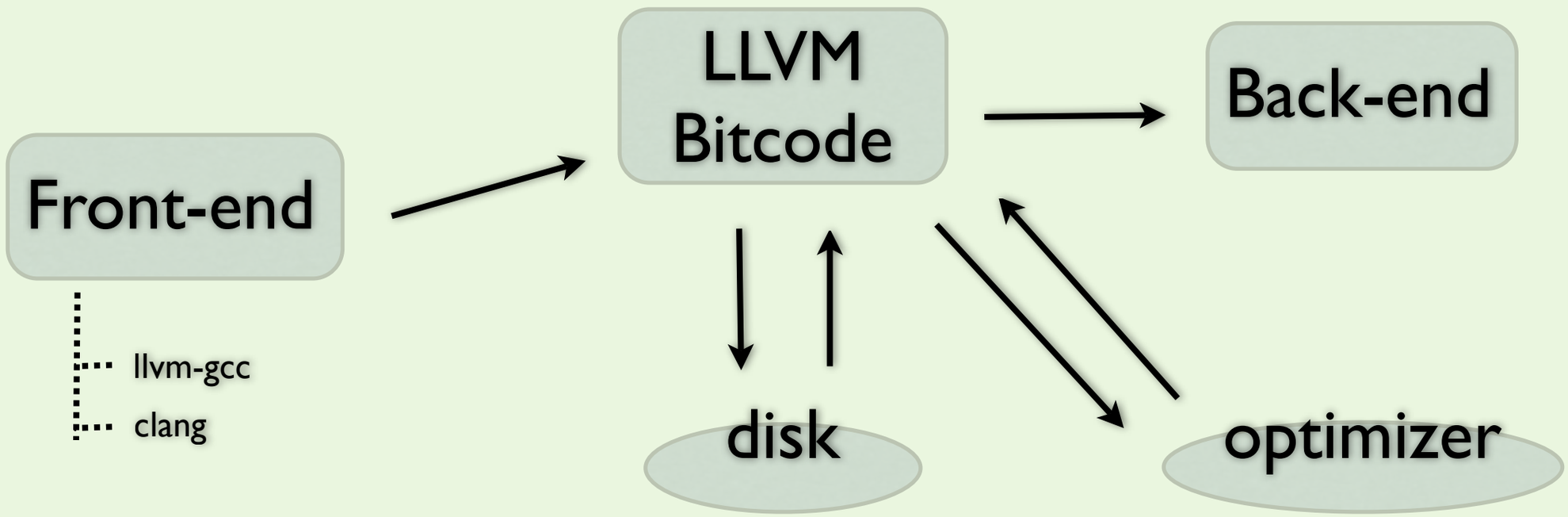
Front-end : clang

```
$ clang -fsyntax-only ~/bozo.c -pedantic
  /tmp/bozo.c:2:17: warning: extension used
  typedef float V __attribute__((vector_size(16)));
                  ^
  1 diagnostic generated.
```

```
$ clang vect.c -emit-llvm | opt -std-compile-opts | llvm-dis

define <4 x float> @foo(<4 x float> %a, <4 x float> %b) {
entry:
  %0 = mul <4 x float> %b, %a
  %1 = add <4 x float> %0, %a
  ret <4 x float> %1
}
```

Optimization oriented compiler

- Provides ***compile time***, ***link-time*** and ***run-time*** optimizations (profile-guided transformations collected by a dynamic profiler).



Optimizations

- Compile-time optimizations
- Driven with -O{1,2,3,s} in llvm-gcc
- Link-time (cross-file, interprocedural) optimizations
- 32 analysis passes and 63 transform passes

Optimizations

-adce	Aggressive Dead Code Elimination
-tailcallelim	Tail Call Elimination
-instcombine	Combine redundant instructions
-deadargelim	Dead Argument Elimination
-aa-eval	Exhaustive Alias Analysis Precision Evaluator
-anders-aa	Andersen's Interprocedural Alias Analysis
-basicaa	Basic Alias Analysis (default AA impl)

LLVM Usage Examples

```
$ clang vect.c -emit-llvm | opt -std-compile-opts | llvm-dis

define <4 x float> @foo(<4 x float> %a, <4 x float> %b) {
entry:
  %0 = mul <4 x float> %b, %a
  %1 = add <4 x float> %0, %a
  ret <4 x float> %1
}
```

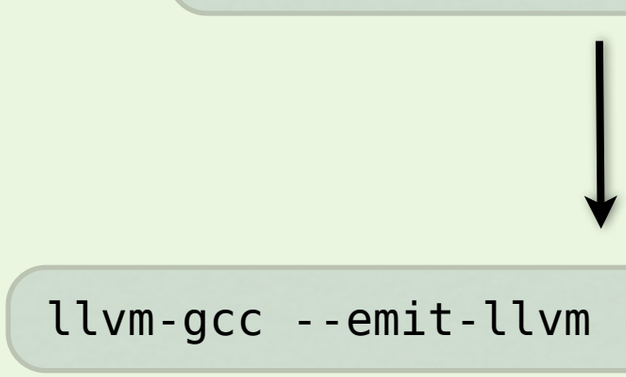
```
$ clang vect.c -emit-llvm | opt -std-compile-opts | llc
  -march=x86 -mcpu=yonah

_foo:
    mulps  %xmm0, %xmm1
    addps  %xmm0, %xmm1
    movaps %xmm1, %xmm0
    ret
```

Putting it all together

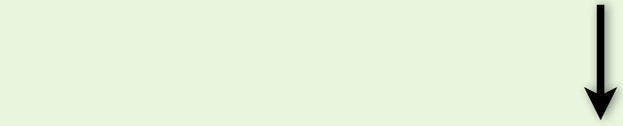
Front-end

Optimizations

```
float bozo(int lhs, int rhs,
           float w)
{
   float c = (lhs/rhs)*w;
   return c;
}
```



```
llvm-gcc --emit-llvm -c bozo.c -o bozo.bc
```



```
llvm-extract -func=bozo bozo.bc |
opt -std-compile-opts
```

Putting it all together

Optimizations

Back-end

```
define float @bozo(i32 %lhs, i32 %rhs, float %w) {
entry:
    %0 = sdiv i32 %lhs, %rhs
    %1 = sitofp i32 %0 to float
    %2 = mul float %1, %w
    ret float %3
}
```

```
... | llc -march=arm
```

```
bozo:
  stmfd sp!, {r4, lr}   mov r1, r4
  sub sp, sp, #8        bl __mulsf3
  mov r4, r2            add sp, sp, #8
  bl __divsi3           ldmfd sp!, {r4, pc}
  bl __floatsisf        .size bozo, .-bozo
```

Why LLVM ?

- Doesn't have a very steep learning curve compared to other known compilers
- Easy to apply custom transformations and optimizations in **anytime** of compilation
- Open source - BSD based license
- Very active community
- Patches get reviewed and integrated to mainline quickly

Design

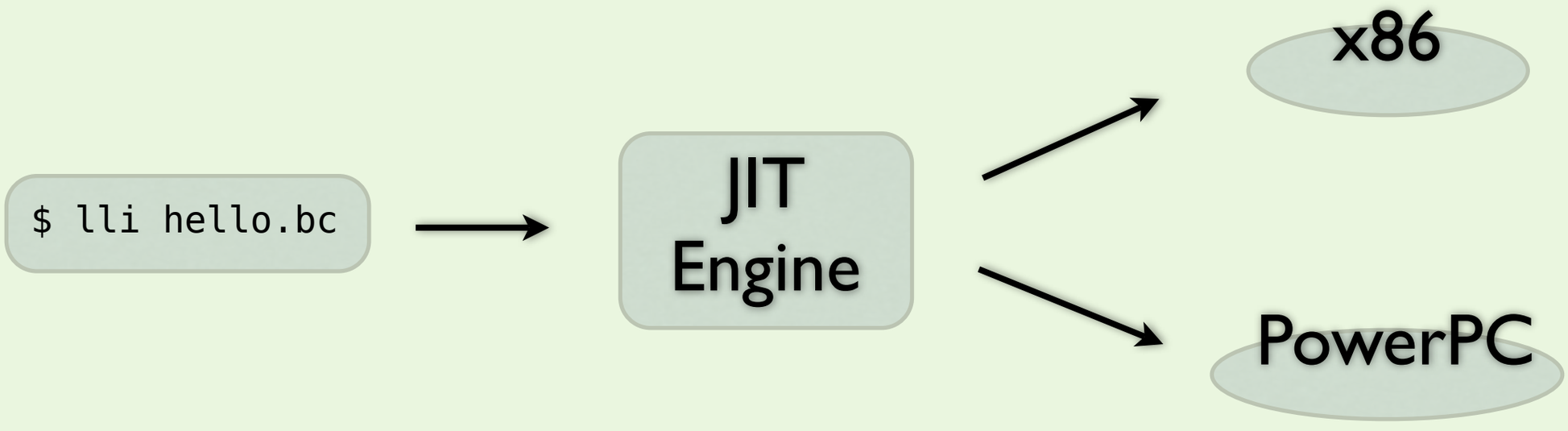
- Well written C++
- Everything implemented as passes
- Easy to plug/unplug transformations and analysis
- Pluggable register allocators
- Modular and Pluggable optimization framework
- Library approach (Runtime, Target, Code Generation, Transformation, Core and Analysis libraries)

The back-end

- Targets:

  Alpha, ARM, C, CellSPU, IA64, Mips, MSIL, PowerPC, Sparc, X86, X86_64, XCore, PIC-16, MSP430, SystemZ, Blackfin

- JIT for X86, X86-64, PowerPC 32/64, ARM



The back-end

- Each back-end is a standalone library.

```
ARMISelDAGToDAG.cpp          ARMInstrVFP.td
ARMJITInfo.cpp               ARMSubtarget.cpp
ARMCodeEmitter.cpp           ARMInstrFormats.td
ARMLoadStoreOptimizer.cpp    ARMConstantIslandPass.cpp
ARMInstrInfo.td              ARMRegisterInfo.h
ARMInstrThumb.td             ARMRegisterInfo.td
```

Back-end tasks

- Codegen
- Support the target ABI
  - Translate the IR to real instructions and registers.
    - Instruction selection
    - Scheduling
    - Target specific optimizations

How's that done?

- LLVM has a **target independent** code generator.
- Inheritance and overloading are used to specify target specific details.
- **TableGen** language, created to describe information and generate C++ code.

TableGen

- TableGen can describe the architecture Calling Convention, Instruction, Registers, ...
- High and low level representations at the same time, e.g. bit fields and DAGs could be represented

```
def RET {    // Instruction MipsInst FR
  field bits<32> Inst = {..., rd{4}, rd{3}, rd{2}, rd{1}, ...};
  dag OutOperandList = (outs);
  dag InOperandList = (ins CPURegs:$target);
  string AsmString = "jr $target";
  list<dag> Pattern = [(MipsRet CPURegs:$target)];
  ...
  bits<6> funct = { 0, 0, 0, 0, 1, 0 };
}
```

Registers

```
def ZERO : MipsGPRReg< 0, "ZERO">, DwarfRegNum<[0]>;
def AT   : MipsGPRReg< 1, "AT">,   DwarfRegNum<[1]>;
def V0   : MipsGPRReg< 2, "2">,    DwarfRegNum<[2]>;
```

Subtargets

ARM

```
  def : Proc<"arm1176jzf-s", [ArchV6, FeatureVFP2]>;
```

PPC

```
  def : Processor<"g4+", G4PlusItineraries, [Directive750, FeatureAltivec]>;
```

Calling Conventions

Describe target specific
ABI information

```
def CC_MipsEABI : CallingConv<[
  // Promote i8/i16 arguments to i32.
  CCIfType<[i8, i16], CCPromoteToType<i32>>,

  // Integer arguments are passed in integer registers.
  CCIfType<[i32], CCAssignToReg<[A0, A1, A2, A3, T0, T1, T2, T3]>>,
  ...
  CCIfType<[f32], CCIfSubtarget<"isNotSingleFloat()",
                  CCAssignToReg<[F12, F14, F16, F18]>>>,

  // Integer values get stored in stack slots that are 4 bytes in
  // size and 4-byte aligned.
  CCIfType<[i32, f32], CCAssignToStack<4, 4>>,
  ...
]>;
```

Legalization and Lowering

- Specify which nodes are legal on the target.
  - Legal nodes can be directly mapped to a target machine instruction.
  - Expansion replaces the node with distinct nodes to represent the same operation.
  - Customization transforms the node into target specific nodes.

Legalization and Lowering

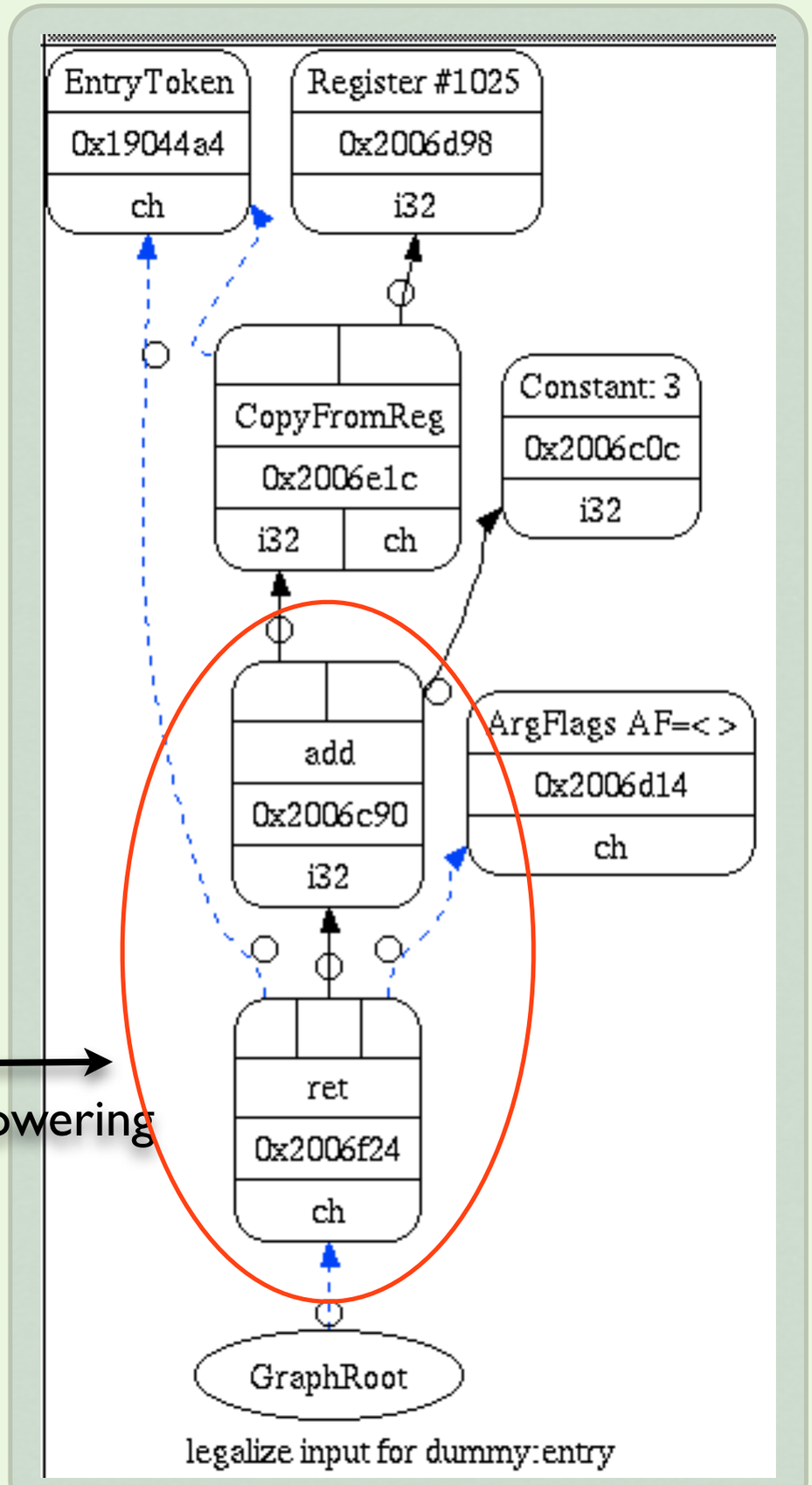
```
setOperationAction(ISD::JumpTable,           MVT::i32, Custom);
setOperationAction(ISD::DYNAMIC_STACKALLOC, MVT::i32, Custom);
setOperationAction(ISD::SELECT,              MVT::i32, Custom);

if (!Subtarget->hasSEInReg()) {
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i8,  Expand);
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i16, Expand);
}
```

DAG before legalization and lowering

```
int dummy(int a) {
  return a+3;
}
```

```
define i32 @dummy(i32 %a) nounwind readnone {
entry:
     %0 = add i32 %a, 3
     ret i32 %0
}
```

Before lowering



Legalization and Lowering

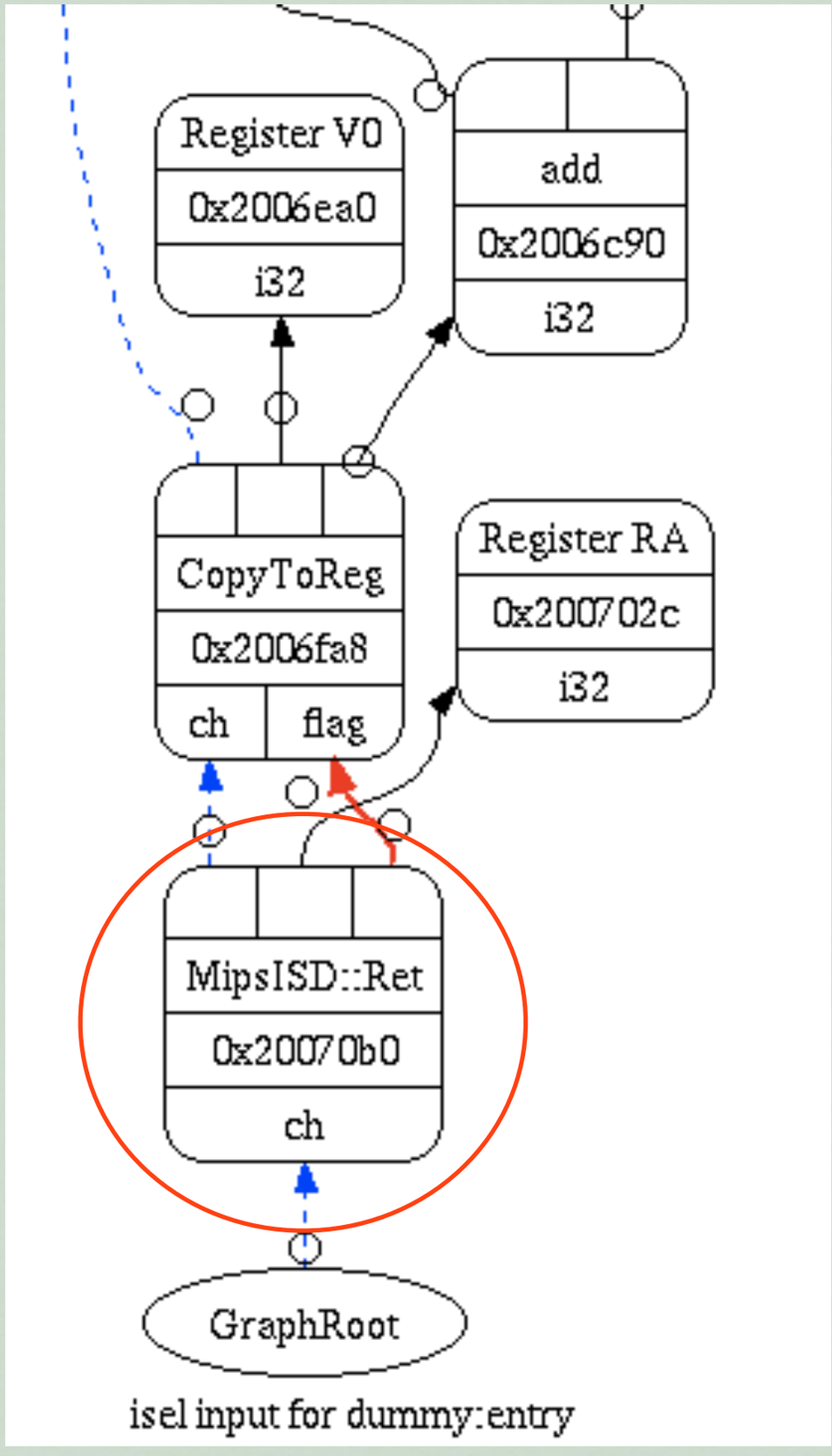
Node Lowering

CALL, FORMAL_ARGUMENTS, Ret,
GlobalAdress, JumpTable


```
LowerOperation(SDOperand Op, SelectionDAG &DAG)
{
  switch (Op.getOpcode())
  {
    case ISD::CALL:                return LowerCALL(Op, DAG);
    case ISD::JumpTable:           return LowerJumpTable(Op, DAG);
    case ISD::GlobalAddress:       return LowerGlobalAddress(Op, DAG);
    case ISD::RET:                 return LowerRET(Op, DAG);
    case ISD::DYNAMIC_STACKALLOC: return LowerDYNAMIC_STACKALLOC(Op, DAG);
    ...
```

DAG lowered

- Some nodes must always be transformed into target specific ones (customized) : CALL, FORMAL_ARGUMENTS, RET



isel input for dummy:entry

Instruction Selection

- After legalization and lowering
- Nodes are matched with target instructions defined by TableGen or handled by special C++ code

Direct instruction matching

```
def ADDiu {
 list<dag> Pattern = [(set CPURegs:$dst,
                           (add CPURegs:$b, immSExt16:$c))];
}
```

C++ code handling

```
case MipsISD::JmpLink: {
      if (TM.getRelocationModel() == Reloc::PIC_) {
      ....
```

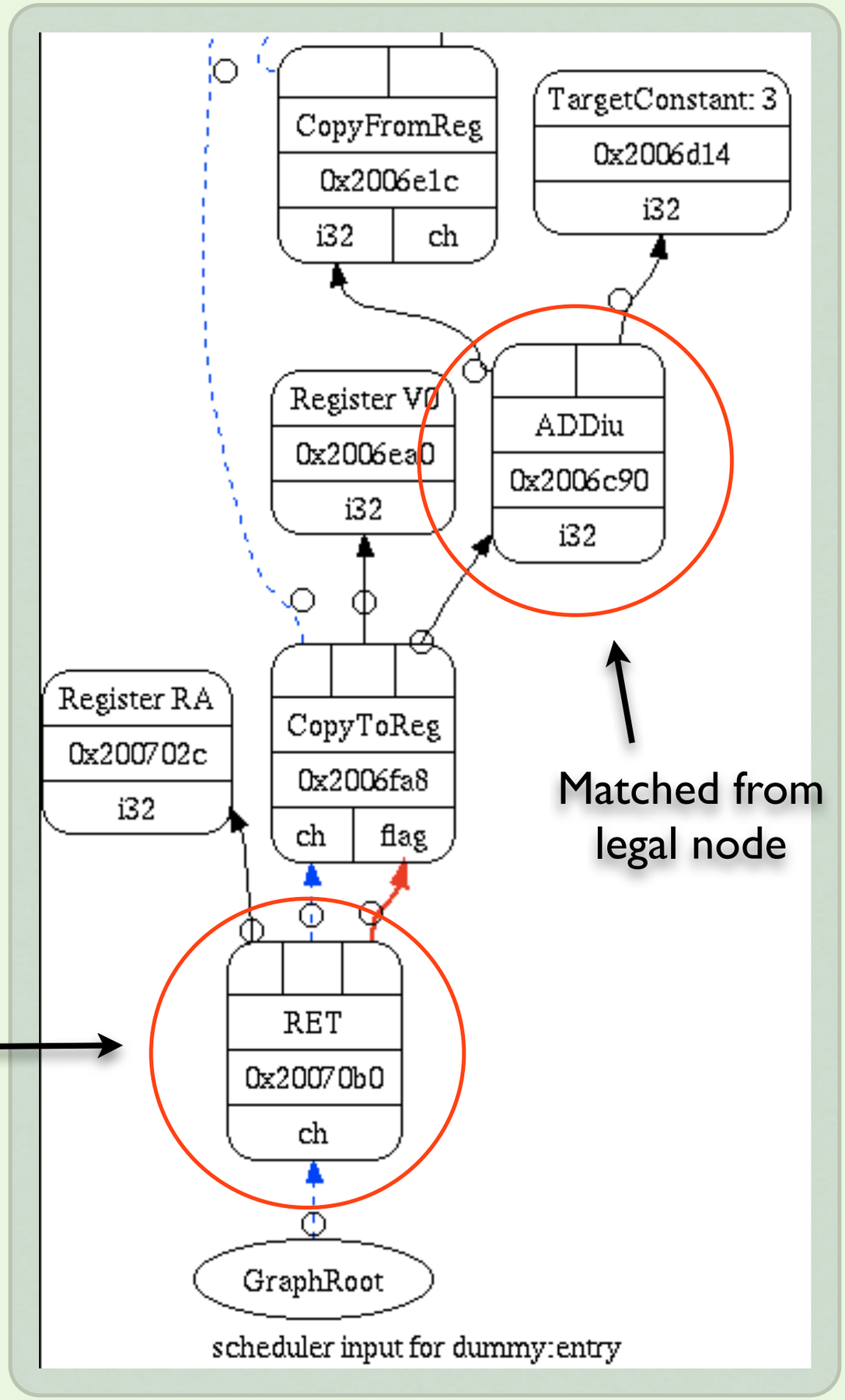
Patterns

- Patterns used to help instruction selection
- Perform peephole optimizations.

```
// Small immediates
def : Pat<(i32 immSExt16:$in),
          (ADDiu ZERO, imm:$in)>;

// Arbitrary immediates
def : Pat<(i32 imm:$imm), (ORi (LUi (HI16 imm:$imm)), (LO16 imm:$imm))>;
```

```
def : Pat<(not CPURegs:$in),
          (NOR CPURegs:$in, ZERO)>;
```

DAG after instruction selection



Target optimizations

- Registered as passes

```
bool ARMTargetMachine::addPreEmitPass(PassManagerBase &PM, bool Fast) {

  if (!Fast && !DisableLdStOpti && !Subtarget.isThumb())
    PM.add(createARMLoadStoreOptimizationPass());

  if (!Fast && !DisableIfConversion && !Subtarget.isThumb())
    PM.add(createIfConverterPass());

  PM.add(createARMConstantIslandPass());
  return true;
}
```

```
bool SparcTargetMachine::addPreEmitPass(PassManagerBase &PM, bool Fast)
{
  PM.add(createSparcFPMoverPass(*this));
  PM.add(createSparcDelaySlotFillerPass(*this));
  return true;
}
```

Mips

- Supports O32 and EABI.
- Mips I and allegrex core (PSP)
- No target specific optimizations.

ARM

- Support several flavors:
  - v4t, v5t, v5te, v6, v6t2, v7a,
- Features:
  - vfp2, vfp3, neon, thumb2

ARM

- Constant Islands pass:
  - ARM specific optimization.
  - Limited PC-relative displacements.
  - Constants are scattered among instructions in a function.

ARM

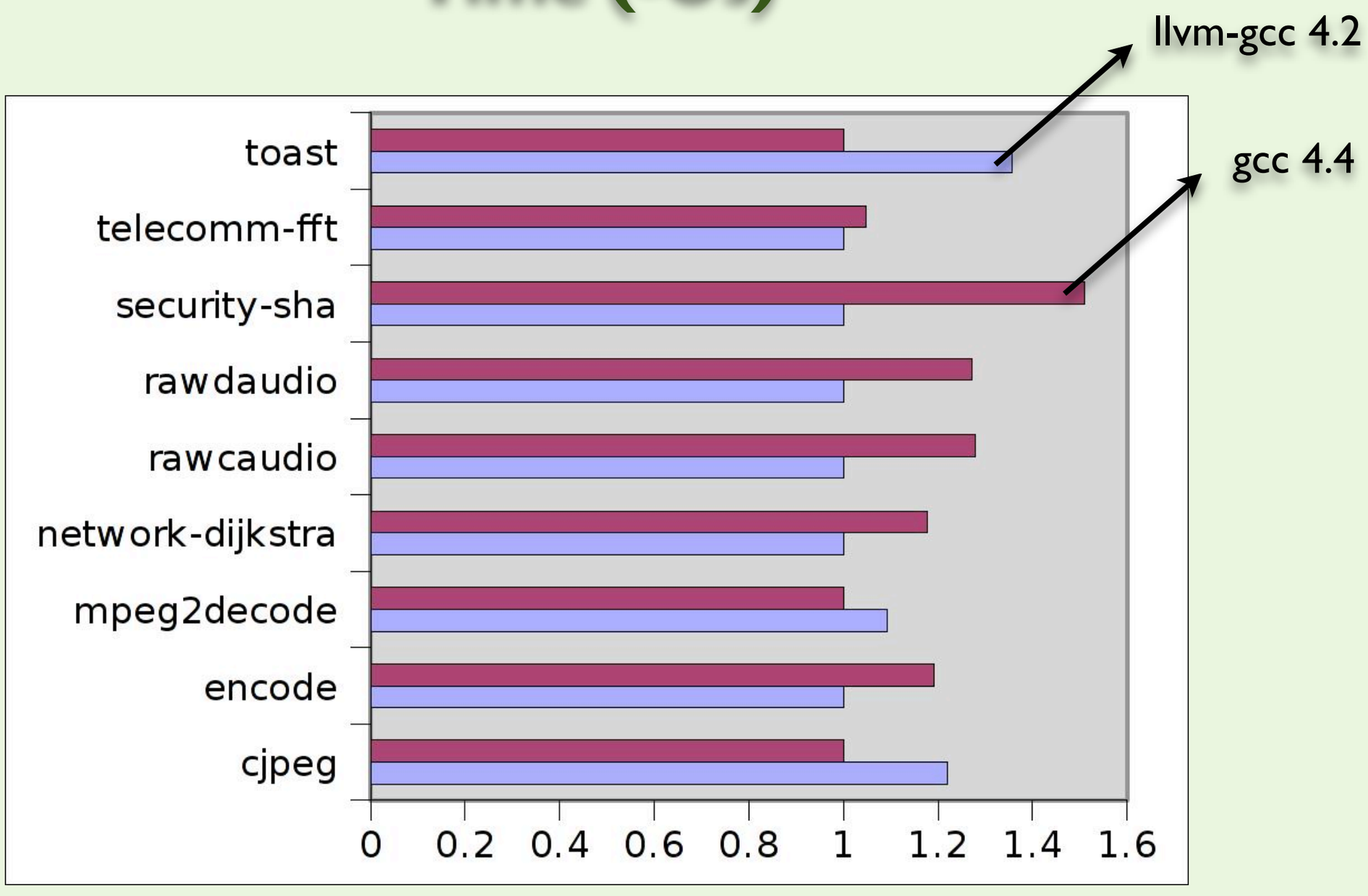
- Load/store optimizer
  - Performs load/store related peephole optimizations
  - Merge several load/store instructions into one or more load/store multiple instructions

ARM

- Thumb2
  - Code size reduction pass
    - 32-bit insns to 16-bit ones
    - 32-bit insns to 2addr 16-bit ones
    - 32-bit load/store to 6-bit ones
    - IT Block pass.

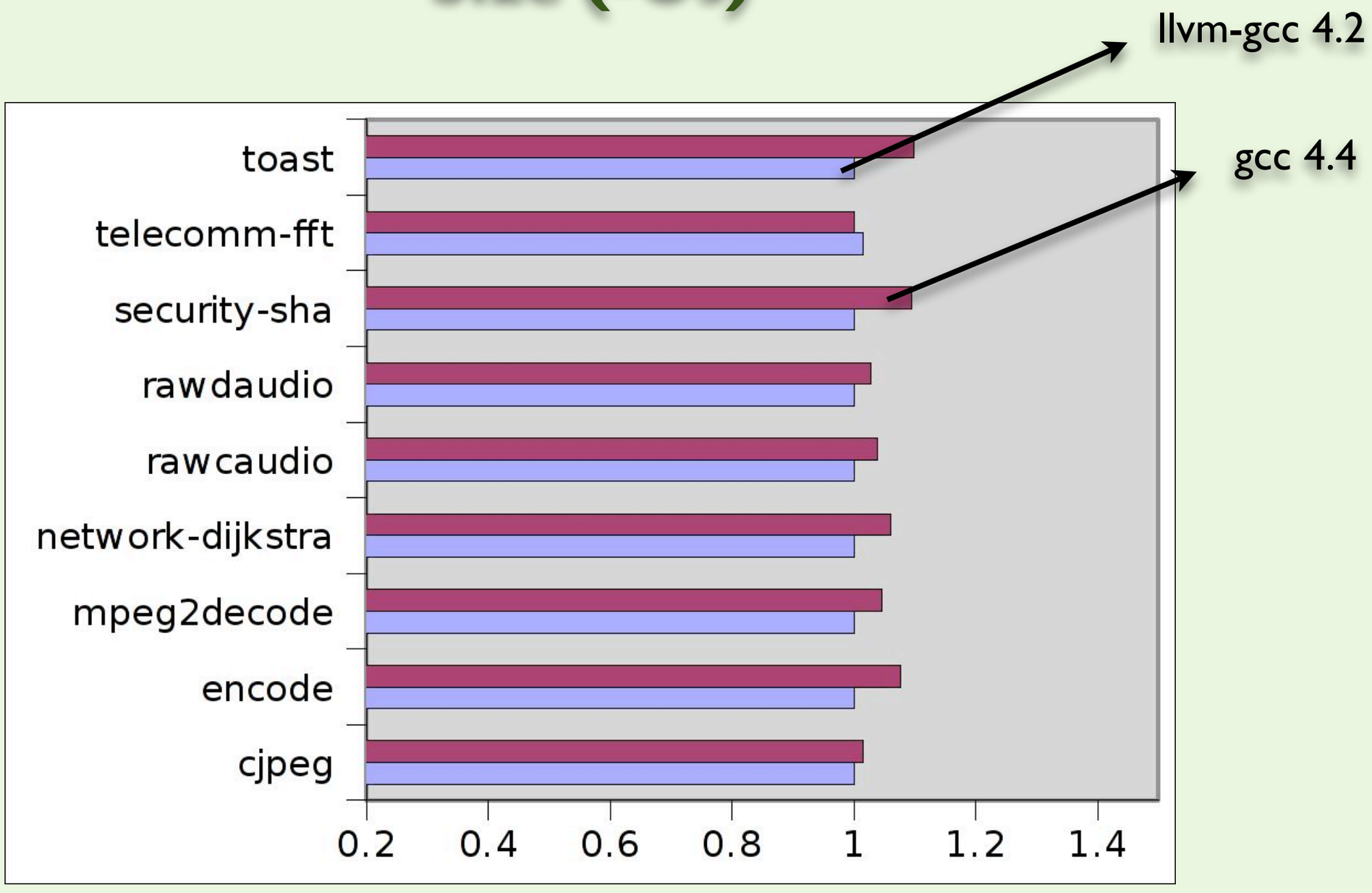
ARM

Time (-Os)


llvm-gcc 4.2
gcc 4.4
toast
telecomm-fft
security-sha
rawdaudio
rawcaudio
network-dijkstra
mpeg2decode
encode
cjpeg
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6

ARM

Size (-Os)


llvm-gcc 4.2
gcc 4.4
toast
telecomm-fft
security-sha
rawdaudio
rawcaudio
network-dijkstra
mpeg2decode
encode
cjpeg
0.2
0.4
0.6
0.8
1
1.2
1.4

The LLVM MIPS and ARM back-end

Embedded Linux Conference 2009
Grenoble, France

Bruno Cardoso Lopes
bruno.cardoso@gmail.com